

Extending FeatherTrait Java with Interfaces

Dedicated to Mario Coppo, Mariangiola Dezani-Ciancaglini, and Simona Ronchi della Rocca on the occasion of their 60th birthday

Luigi Liquori^a and Arnaud Spiwack^b

^a*INRIA, France*

^b*ENS Cachan, France*

Abstract

In the context of Featherweight Java by Igarashi, Pierce, and Wadler, and its recent extension FeatherTrait Java (FTJ) by the authors, we investigate classes that can be extended with trait composition. A trait is a collection of methods, *i.e.*, behaviors without state; it can be viewed as an “incomplete stateless class” *i.e.*, an interface with some already written behavior. Traits can be composed in any order, but only make sense when “imported” by a class that provides state variables and additional methods to disambiguate conflicting names arising between the imported traits. We introduce FeatherTrait Java with Interfaces (iFTJ), where traits need to be typechecked only once, which is necessary for compiling them in isolation, and considering them as regular types, like Java-interfaces with a behavioral content.

Key words: Object-oriented language design, inheritance, types.

1 Introduction

Untyped Traits, introduced by Schärli, Ducasse, Nierstrasz, Wuyts, and Black [5, 13, 16, 17], have recently emerged as a novel technique for building composable units of behavior in a dynamically-typed language *à la* Smalltalk. Intuitively, a trait is just a collection of methods, *i.e.*, behavior *without* state. Derived traits can be built from an *unordered* list of parent traits, together with new method declarations. Thus, traits are (incomplete) classes without state. Traits can be composed in any order. A trait makes sense only when “imported” by a class that provides state variables and possibly additional methods to disambiguate conflicting names arising among the imported traits. The order for importing traits in classes is irrelevant.

Historically, traits, intended as a collection of state *and* behavior, have been originally employed in the pure object-based languages Self [20], or in the language Obliq [4], or for the encoding of classes as records-of-premethods in the Abadi-Cardelli’s Object Calculus [1].

Typed traits, intended as pure behavior *without* state *à la* Schärli *et al.* [5, 16, 17], have been introduced by Fisher and Reppy in an object-based core calculus for the **Moby** language (of the ML [11] family) [7, 12]. Then, traits have been immersed in Igarashi, Pierce, and Wadler **Featherweight Java** by Liquori and Spiwack [10], studied by Smith and Drossopoulou in a **Java** setting [18], and implemented by Odersky *et al.* in the class-based language **Scala** [15], and in the new language **Fortress** by Allen, Chase, Luchangco, Maessen, Ryu, Steele, and Tobin-Hochstadt [2].

Contributions. The starting point of this paper is the **FeatherTrait Java** (FTJ) calculus, by Liquori and Spiwack [10], that conservatively extends the simple calculus of **Featherweight Java** (FJ) by Igarashi, Pierce, and Wadler [9] with statically typed traits. The main aim of FTJ was to introduce a typed trait-based inheritance in a class-based calculus *à la Java*. Because of the simplicity of the FTJ calculus, traits could be typechecked only *inside a class*, thus they needed to be typechecked *once for every class*. This behavior is not compatible with the idea of compiling traits in isolation. In iFTJ, the traits need only be typechecked *once and for all*.

- (1) We define the **FeatherTrait Java with Interfaces** calculus (iFTJ), a variation of FTJ and a conservative extension of FJ, which allows traits to be typechecked only once. Traits in iFTJ look like **Java**-interfaces with some partial behavior inside. An example of what traits can look like is

```
trait TA {String p(){return this.r()+this.s()+this.q();}
         String s(){return "Java";};
         String r()
         String q()}
trait TB {String r(){return "Hallo World, my name is";}
         String s(){return "FeatherTrait Java";};}
```

Traits TA and TB are typechecked only once, thus could be compiled in isolation; trait TA “defines” method `s`, and method `p` which “requires” methods `r` and `q` (declared as interfaces). They can be both imported in a class declaration as follows

```
class Presentation extends Object imports TA TB
{;Presentation(){super();}
  String ciao(){return this.p();}
  String   s(){return "FeatherTrait Java with Interfaces,";};
  String   q(){return "I hope you will like me";};}
```

Multiple traits can be imported by one class, and conflicts between common methods, defined in two or more inherited traits, must be resolved *explicitly* by the user, either by aliasing or excluding method names in traits, or by overriding the conflicted methods in the class that imports those traits or in the trait itself. As such, the evaluation of `(new Presentation()).ciao()` will produce “Hallo World, my name is FeatherTrait Java with Interfaces, I hope you will like me”.

- (2) We define a type system for iFTJ that typechecks traits only once, in order to be compatible with compilation in isolation. In a nutshell, every trait is typechecked using a judgment which lists the signatures of methods that are *required* in order to *complete* the missing behavior of the trait itself.

Outline of the paper. The paper is structured as follows. In Section 2, we quickly review the trait inheritance model adopted in iFTJ. In Section 3, we present the syntax, the operational semantics, and the type system of iFTJ. Section 4 shows the type soundness of iFTJ. Section 5 presents an example of using traits in iFTJ. Section 6 discusses related work and concludes. Appendix A sums up the operational semantics and the type system of iFTJ. Appendix B contains the detailed soundness proof for iFTJ. Because of a lack of space in this volume, a longer version can be found on the authors' web pages.

2 Trait Inheritance

One useful feature of *trait-based inheritance* is that when a conflict arises between traits included in the same class (*e.g.*, a method defined in two different traits), then the conflict is signaled and it is up to the user to *explicitly* and *manually* resolve the conflict. Three simple rules can be easily implemented in the method-lookup algorithm for that purpose

- (1) Methods defined in a class take precedence over methods defined in the traits imported by the class.
- (2) Methods defined in a composite trait take precedence over methods defined in the imported traits.
- (3) Methods defined in traits (imported by a class) take precedence over methods defined in its parent class.

The above rules are the simple recipe of the trait-based inheritance model. They greatly increase the flexibility of the calculus that uses traits. Traits syntactically require the methods which are necessary to “complete” their behavior. They can also import other traits, from which they gain both implemented and required methods.

Conflict Resolution. When dealing with trait inheritance, conflicts may arise; a class C might import two traits T_1 and T_2 defining the same method p with different behavior. Conflicts between traits must be resolved *manually*, *i.e.*, there is no special or rigid discipline to learn how to use traits. Once a conflict is detected, there are essentially three ways to resolve the conflict

- (1) **Overriding a new method p inside the class.** A new method p is redefined inside the class with a new behavior. The (trait-based) lookup

algorithm will hide the conflict in the traits in favor of the overridden method defined in the class.

```
class C extends Object imports T1 T2          T1 and T2 both define p
{... D p(...){...}}                          new behavior for p
```

- (2) **Aliasing the method p in traits and redefining the method in the class.** The method p is aliased with new different names. A new behavior for p can now be given in the class C (possibly re-using the aliased methods p_of_T1 and p_of_T2 which are no longer conflicting).

```
class C extends Object imports
T1 with {p@p_of_T1}                          aliases p with p_of_T1
T2 with {p@p_of_T2}                          aliases p with p_of_T2
{... D p(...){...}}                          new behavior for p, it may use p_of_T1/2
```

- (3) **Excluding the method p in one of the traits.** One method p in trait $T1$ or $T2$ is excluded. This solves the conflict in favor of one trait.

```
class C extends Object imports
T1 (T2 minus {p}) {...}                       method p is hidden from T2
```

3 FeatherTrait Java with Interfaces

In iFTJ, a program consists of a collection of class and trait declarations, and an expression to be evaluated. We adopt the same notational conventions and hygiene conditions as the FJ paper [9], with the metavariables S and I , ranging over *signatures* and types, respectively, and the metavariables $M_{\perp}, S_{\perp}, (\bar{x}, e)_{\perp}$ ranging over methods (resp. signatures and method bodies) and the special failure value *fail*.

CL	::=	class C extends C [imports \overline{TA}]{ $\overline{I} \overline{f}; K \overline{M}$ }	Class Declarations
TL	::=	trait T [imports \overline{TA}]{ $\overline{M}; \overline{S}$ }	Trait Declarations
I	::=	C T	Types
TA	::=	T TA with { $m@m$ } TA minus { m }	Trait Alterations
K	::=	C($\overline{I} \overline{f}$){super(\overline{f}); this. $\overline{f} = \overline{f}$; }	Constructors
M	::=	I m($\overline{I} \overline{x}$){return e; }	Methods
S	::=	I m($\overline{I} \overline{x}$)	Signatures
e	::=	x e.f e.m(\overline{e}) new C(\overline{e}) (I)e	Expressions

Figure 1. Syntax of iFTJ

3.1 Syntax and Operational Semantics

The syntax of iFTJ, presented in Figure 1, extends the syntax of FJ. An iFTJ program is a triple (CT, TT, e) of a class and a trait table, and an expression.

A class `class C extends C imports1 \overline{TA} { \overline{I} \overline{f} ; K \overline{M} }` in iFTJ is composed of field declarations \overline{I} \overline{f} , a constructor K , some new or redefined methods \overline{M} , plus a list of imported, possibly altered, traits \overline{TA} . A trait `trait T imports \overline{TA} { \overline{M} ; \overline{S} }` is composed of a list of methods \overline{M} , some other, possibly altered, traits \overline{TA} imported by the trait itself, and a list of abstract method signatures \overline{S} , which are the methods that aren't implemented in the trait but are yet required by it. The conflicts are handled by typechecking: all the conflicts must be resolved manually by the program. If any is found during typechecking, then the program is rejected. The well-known Snyder's "diamond problem" [19] is not considered as a conflict, *i.e.* if two traits T_1 , T_2 inherit a method m from the same trait T_0 , and a trait imports both T_1 and T_2 , then the method m from T_1 and the method m from T_2 will not be considered as conflicting as they are both exactly the same. Expressions are the usual ones of FJ. The subtyping rules are essentially the same as those of FJ plus the two rules:

$$\frac{\overline{TA} \in \overline{TA}}{\text{trait } T \text{ imports } \overline{TA} \{ \dots \}}_{(\text{Sub}\cdot\text{Tr})} \quad \frac{\overline{TA} \in \overline{TA}}{\text{class } C \text{ extends } D \text{ imports } \overline{TA} \{ \dots \}}_{(\text{Sub}\cdot\text{Cla}\cdot\text{Tr})}$$

$$T <: \text{head}(TA) \qquad C <: \text{head}(TA)$$

The subtyping relation does not only compare classes but also traits. To give an intuition about the above rules, we will remind that **Java** typing and subtyping is *name-based* (a type is the name of a class or an interface). We intend to stick to this policy in iFTJ. The function *head* returns the name of the trait which is the *head* of the trait alteration. The rationale is that the alterations do indeed alter the behavioral content of traits, but they do not change their interface²; another point of view is that alterations transform implemented methods into required methods, both being identified at type level. For instance, an object which inhabits `class C imports (T minus {m}){...}` does qualify as being also of type T . The other subtyping rules, the simple definition of *head* and the other standard definition of the operational semantics of iFTJ are collected in Appendix A.

3.2 A Virtual Tour Through the Auxiliary Functions

The Functions *meth* and *sig*. The function *meth* has two purposes. The first one, simpler, is to extract the names from either a method declaration (with a body), or a method signature (without a body); it is used in the rules to convert sets of declaration into sets of names. The second purpose is to compute the set of all methods in a class or a trait or a trait alteration which has an *available*, *real* implementation, not simply a typed interface. Note that the required methods of a trait or trait alteration are not considered by this function. The function *sig* simply extract the set of the signatures of every method (both implemented and required ones) of a trait or a trait alteration. Both functions are presented in Appendix A.

¹ The keyword `imports` was preferred to the keyword `implements` (*à la Java*) because traits already implements some methods.

² The rigid type discipline makes iFTJ a proper extension of FJ but not of FTJ.

The Functions *altlook* and *tlook*. The function *altlook* looks up a trait alteration for the complete implementation of a method m (or fails if it has none, even if there is a declared signature). *altlook* is not a function but it becomes a function for well-typed programs. The function *tlook* is the extension of *altlook* to a set of trait alterations; these two “functions” are mutually recursive. *tlook* is used to find a method in a set of trait alterations; it has no specific strategy to select among multiply defined methods, this is why it is not a function (and subsequently why *altlook* is not a function either). However, typing prevents conflicts, turning both into functions. The most significant rules of *altlook* are

$$\frac{\text{TT}(\text{T}) = \text{trait T imports } \overline{\text{TA}} \{ \overline{\text{M}}; \overline{\text{S}} \} \quad m \notin \text{meth}(\overline{\text{M}}) \quad tlook(m, \overline{\text{TA}}) = M_{\perp}}{altlook(m, \text{T}) = M_{\perp}} \text{(ATr-Inh)}$$

$$\frac{altlook(n, \text{TA}) = I \ n(\overline{\text{I}} \ \overline{x}) \{ \text{return } e; \}}{altlook(m, \text{TA with } \{ n @ m \}) = I \ m(\overline{\text{I}} \ \overline{x}) \{ \text{return } e; \}} \text{(ATr-Ali}_1\text{)}$$

(ATr-Inh) If the method m is not provided in the unaltered trait definition, then we look in the imported traits.

(ATr-Ali₁) When looking up a method m in a trait alteration where n is aliased to m , we look up for the method with the former name n , and then we rename it if it exists, or the lookup fails.

The Function *msig*. The function *msig* looks up a trait alteration (similarly to *altlook*) for a method signature S , or fails (in the case where no signature is found, or the method has an available body). When a method m is *required* by the trait alteration TA , then *msig*(m, TA) returns the *signature* with which m should be (later) implemented in a class. Note that *msig* is not a function in general, but gets to be one in the case of well-typed programs. The most significant rules of *msig* are

$$\frac{\text{TT}(\text{T}) = \text{trait T imports } \overline{\text{TA}} \{ \overline{\text{M}}; \overline{\text{S}} \} \quad m \notin \text{meth}(\text{T}) \quad \exists \text{TA} \in \overline{\text{TA}}. \text{msig}(m, \text{TA}) = S \quad m \notin \text{meth}(\overline{\text{S}})}{msig(m, \text{T}) = S} \text{(MSig-Inh)}$$

$$\frac{\text{TT}(\text{T}) = \text{trait T imports } \overline{\text{TA}} \{ \overline{\text{M}}; \overline{\text{S}} \} \quad m \notin \text{meth}(\text{T}) \quad \forall \text{TA} \in \overline{\text{TA}}. \text{msig}(m, \text{TA}) = \text{fail} \quad m \notin \text{meth}(\overline{\text{S}})}{msig(m, \text{T}) = \text{fail}} \text{(MSig-End)}$$

$$\frac{m \neq n \quad altlook(m, \text{TA}) = I \ m(\overline{\text{I}} \ \overline{x}) \{ \text{return } e; \}}{msig(m, \text{TA with } \{ m @ n \}) = I \ m(\overline{\text{I}} \ \overline{x})} \text{(MSig-Ali}_3\text{)}$$

(MSig-Inh) & (MSig-End) When looking up a required method signature in an unaltered trait, if the method is not spoken of in the trait, then we look in the imported traits. If none of them requires it, then the lookup fails. Note that we ensure that the method is not implemented in any of the imported traits.

(MSig-Ali₃) Those rules apply when looking up a required method m signature in a trait alteration where m is aliased to n . We look whether or not m exists in

the head trait alteration with an implementation. If it does, then it becomes required (we do not change the type interface of the trait alteration even through alterations), otherwise the lookup fails.

The Function *mtype*. As above, *mtype* is a function only for well-typed programs. It fetches in a class or in a trait alteration the type of m which can be either implemented or required. It is used in the typing expressions like $e.m(\dots)$. Whether m is required or implemented does not matter. Concrete objects (`new ...`) are instances of a class; type soundness ensures that classes implement all methods required by the traits they import. As far as the typing of expressions is concerned, traits are like interfaces. The most significant rules of *mtype* are

$$\frac{\text{CT}(\mathbf{C}) = \text{class } \mathbf{C} \text{ extends } \mathbf{D} \text{ imports } \overline{\mathbf{T}\mathbf{A}} \{ \overline{\mathbf{J}} \overline{\mathbf{f}}; \mathbf{K} \overline{\mathbf{M}} \} \\ m \notin \text{meth}(\overline{\mathbf{M}}) \quad \exists \mathbf{T}\mathbf{A} \in \overline{\mathbf{T}\mathbf{A}}. \text{mtype}(m, \mathbf{T}\mathbf{A}) = \overline{\mathbf{I}} \rightarrow \mathbf{I}}{\text{mtype}(m, \mathbf{C}) = \overline{\mathbf{I}} \rightarrow \mathbf{I}} \text{(MTyp}\cdot\text{Tr)}$$

$$\frac{\text{CT}(\mathbf{C}) = \text{class } \mathbf{C} \text{ extends } \mathbf{D} \text{ imports } \overline{\mathbf{T}\mathbf{A}} \{ \overline{\mathbf{J}} \overline{\mathbf{f}}; \mathbf{K} \overline{\mathbf{M}} \} \\ m \notin \text{meth}(\overline{\mathbf{M}}) \quad \forall \mathbf{T}\mathbf{A} \in \overline{\mathbf{T}\mathbf{A}}. \text{mtype}(m, \mathbf{T}\mathbf{A}) = \text{fail} \quad \text{mtype}(m, \mathbf{D}) = \overline{\mathbf{I}} \rightarrow \mathbf{I}}{\text{mtype}(m, \mathbf{C}) = \overline{\mathbf{I}} \rightarrow \mathbf{I}} \text{(MTyp}\cdot\text{Super)}$$

$$\frac{\text{altlook}(m, \mathbf{T}\mathbf{A}) = \text{fail} \quad \text{msig}(m, \mathbf{T}\mathbf{A}) = \mathbf{I} \ m(\overline{\mathbf{I}} \ \overline{\mathbf{x}})}{\text{mtype}(m, \mathbf{T}\mathbf{A}) = \overline{\mathbf{I}} \rightarrow \mathbf{I}} \text{(MTyp}\cdot\text{Virt)}$$

(MTyp·Tr) If a class has not declared a method explicitly, then we first lookup the method type inside the imported traits.

(MTyp·Super) If the method is not declared in the imported traits (either implemented or required), then we lookup inside the superclass.

(MTyp·Virt) This rule applies when looking up for a method type in a trait alteration. If the method is not implemented in the trait alteration, then we look whether it is required by the trait alteration, giving the appropriate type. Thus, traits behave more like interfaces than classes.

The Functions *fields* and *mbody*. Those functions are almost unmodified since FJ. The function *fields* simply computes the set of the fields of a class (this includes those of the superclass) together with their types. *mbody* is a function only for well-typed programs. The function *mbody* performs the method body lookup: given a method m and a class \mathbf{C} , it browses the inheritance tree of \mathbf{C} until it finds the body of m . The most significant rules of *mbody* are

$$\frac{\text{CT}(\mathbf{C}) = \text{class } \mathbf{C} \text{ extends } \mathbf{D} \text{ imports } \overline{\mathbf{T}\mathbf{A}} \{ \overline{\mathbf{I}} \overline{\mathbf{f}}; \mathbf{K} \overline{\mathbf{M}} \} \\ m \notin \text{meth}(\overline{\mathbf{M}}) \quad \text{tlook}(m, \overline{\mathbf{T}\mathbf{A}}) = \mathbf{I} \ m(\overline{\mathbf{I}} \ \overline{\mathbf{x}}) \{ \text{return } \mathbf{e}; \}}{\text{mbody}(m, \mathbf{C}) = (\overline{\mathbf{x}}, \mathbf{e})} \text{(MBdy}\cdot\text{Tr)}$$

$$\frac{\text{CT}(\mathbf{C}) = \text{class } \mathbf{C} \text{ extends } \mathbf{D} \text{ imports } \overline{\mathbf{T}} \{ \overline{\mathbf{I}} \overline{\mathbf{f}}; \mathbf{K} \overline{\mathbf{M}} \} \\ m \notin \text{meth}(\overline{\mathbf{M}}) \quad \text{tlook}(m, \overline{\mathbf{T}\mathbf{A}}) = \text{fail} \quad \text{mbody}(m, \mathbf{D}) = (\overline{\mathbf{x}}, \mathbf{e})_{\perp}}{\text{mbody}(m, \mathbf{C}) = (\overline{\mathbf{x}}, \mathbf{e})_{\perp}} \text{(MBdy}\cdot\text{SCla)}$$

(MBdy·Tr) If the method is not declared in the class, then we first look it up in the imported traits.

(MBdy·SCla) If the method is not in any of the imported traits, then we look it up in the superclass.

Method Path relation \triangleleft . This relation is related to “diamond” (or “fork-join”) conflicts arising when a class/trait, that inherits from two classes/traits, would ostensibly have two distinct definitions for one method [19]. The set $\cap\overline{\text{TA}}$ denotes methods defined in more than one trait; it is used to detect conflicts when importing traits. The set $\diamond\overline{\text{TA}}$ denotes methods that potentially determine a diamond when dealing with trait inheritance; such methods are expected to be “non-conflicting”, hence accepted by the type system. More precisely: the set $\cap\overline{\text{TA}}$ detects every conflict in $\overline{\text{TA}}$, while the set $\diamond\overline{\text{TA}}$ detects every diamond. A class declaration is well-formed only if the imported trait alterations imported by the class \mathbf{C} satisfy the constraint $\cap\overline{\text{TA}} \setminus \diamond\overline{\text{TA}} \subseteq \text{meth}(\overline{\mathbf{M}})$, ensuring that every conflict is resolved, *i.e.*, every new-born conflict ($\cap\overline{\text{TA}}$) which is not a diamond ($\diamond\overline{\text{TA}}$) is being overridden.

$$\cap\overline{\text{TA}} \stackrel{\text{def}}{=} \{m \mid \exists \text{TA}_1 \neq \text{TA}_2 \in \overline{\text{TA}}. m \in \text{meth}(\text{TA}_1) \cap \text{meth}(\text{TA}_2)\}$$

$$\diamond\overline{\text{TA}} \stackrel{\text{def}}{=} \{m \mid \exists n, \text{TA}_1. \forall \text{TA}_2 \in \overline{\text{TA}}. m \in \text{meth}(\text{TA}_2) \implies m \text{ in } \text{TA}_2 \triangleleft n \text{ in } \text{TA}_1\}$$

To compute $\diamond\overline{\text{TA}}$, we need a judgment of the form $m \text{ in } \text{TA}_1 \triangleleft n \text{ in } \text{TA}_2$ (read “ m of TA_1 behaves exactly as n of TA_2 ”). The meaning is as follows: m is a method provided by trait TA_1 , whose implementation is inherited from that of a method n provided by TA_2 through any number of trait declarations or alteration steps (paths). The most significant rules are

$$\begin{array}{c} \text{TT}(\text{T})=\text{trait } \text{T} \text{ imports } \overline{\text{TA}} \{ \overline{\mathbf{M}}; \overline{\mathbf{S}} \} \\ \hline \text{TA} \in \overline{\text{TA}} \quad m \in \text{meth}(\text{TA}) \setminus \text{meth}(\overline{\mathbf{M}}) \quad \frac{p \text{ in } \text{TA}_1 \triangleleft n \text{ in } \text{TA}_2}{m \text{ in } \text{TA}_1 \text{ with } \{p@m\} \triangleleft n \text{ in } \text{TA}_2} \text{(Path}\cdot\text{Ali}_1) \\ \hline m \text{ in } \text{T} \triangleleft m \text{ in } \text{TA} \end{array}$$

$$\frac{m \text{ in } \text{TA}_1 \triangleleft n \text{ in } \text{TA}_2 \quad m \neq p \quad m \neq q}{m \text{ in } \text{TA}_1 \text{ with } \{p@q\} \triangleleft n \text{ in } \text{TA}_2} \text{(Path}\cdot\text{Ali}_2) \quad \frac{m \text{ in } \text{TA}_1 \triangleleft n \text{ in } \text{TA}_2 \quad m \neq p}{m \text{ in } \text{TA}_1 \text{ minus } \{p\} \triangleleft n \text{ in } \text{TA}_2} \text{(Path}\cdot\text{Exl)}$$

- (Path·Inh) If a trait T inherits a method m directly from a trait alteration TA and does not override it, then m of T behaves exactly as m of TA .
- (Path·Ali₁) If p of TA_1 behaves exactly as n of TA_2 , then m of TA_1 with $\{p@m\}$ behaves exactly as n of TA_2 .
- (Path·Ali₂) If $m \neq p$ and $m \neq q$, and m of TA_1 behaves exactly as n of TA_2 , then m of TA_1 with $\{p@q\}$ behaves exactly as n of TA_2 .
- (Path·Exl) If $m \neq p$, and m of TA_1 behaves exactly as n of TA_2 , then m of TA_1 minus $\{p\}$ behaves exactly as n of TA_2 .

3.3 The Type System

We show the most important rules of iFTJ type system which allow to typecheck traits only once. The remaining rules are presented in Appendix A. The type system has three steps: first, expressions must be typed using

standard judgments of the form $\Gamma \vdash e \in I$. Second, methods must be typed inside a class using judgments of the form $M \text{ OK IN } C$ or inside a trait. Since a trait is essentially a **Java**-interface with some behavior inside, it has a type of its own. The associated judgment $M \text{ OK IN } T$ (and typing rules) are similar. Judgments $S \text{ OK IN } T$ and $S \text{ OK IN } C$ hold to guarantee that signatures are compatible. Next, altered traits must be typed using judgments of the form $TA \text{ OK requires } \bar{S}$. The intuition behind the **requires** \bar{S} part is that the implementation of the \bar{S} methods must be available in the classes which import TA with the given signature. The implementation may be given either by an explicit declaration in the body of the class (or trait), or inherited by the superclass or by another trait. Signatures are considered equal modulo renaming of their arguments. Finally, trait and class typechecking are performed only once. Checking classes and traits is done via judgments of the form $TL \text{ OK requires } \bar{S}$ and $CL \text{ OK}$ where the trait and class tables TT and CT are left implicit in the judgments. Like trait alterations, trait declaration checking gives also the signature of the abstract methods.

Method typechecking is defined as follows

$$\begin{array}{c}
 CT(C) = \text{class } C \text{ extends } D \text{ imports } \bar{TA} \{ \bar{J} \bar{f}; K \bar{M} \} \\
 \bar{x}:\bar{I}, \text{this}:C \vdash e \in J \quad J <: I \\
 \text{override}(m, D, \bar{I} \rightarrow I) \quad \text{override}(m, \bar{TA}, \bar{I} \rightarrow I) \\
 \hline
 I \ m(\bar{I} \ \bar{x})\{\text{return } e;\} \text{ OK IN } C \quad \text{(Mth-Ok-Cla)} \\
 \\
 TT(T) = \text{trait } T \text{ imports } \bar{TA} \{ \bar{M}; \bar{S} \} \\
 \bar{x}:\bar{I}, \text{this}:T \vdash e \in J \quad J <: I \quad \text{override}(m, \bar{TA}, \bar{I} \rightarrow I) \\
 \hline
 I \ m(\bar{I} \ \bar{x})\{\text{return } e;\} \text{ OK IN } T \quad \text{(Mth-Ok-Tr)}
 \end{array}$$

(Mth-Ok-Cla) We first ensure that the method body e is typable with a type compatible with its declared signature, *i.e.* that if the method body has type J , then J is smaller (possibly equal) than the declared type I . We then check the two override conditions, *i.e.* we check that if the method name m is used in any of the imported trait or in the superclass, then it is used with the same type as this method.

(Mth-Ok-Tr) This rule behaves as the previous (Mth-Ok-Cla) rule; it is interesting to remark that the type assigned to the pseudovariable **this** is the trait T itself, which is considered as a *real type*.

Simpler rules apply to check method signatures in a trait or in a class.

The trait alteration typing. These rules derive judgments of the form $TA \text{ OK requires } \bar{S}$ which means that TA is well-typed where every method declared in the signature \bar{S} must be implemented. The rationale is that every method occurring in the require part refers to a method that is not (or no more) implemented in the trait but is *needed in order to complete the trait*.

$$\begin{array}{c}
\text{TA OK requires } \bar{S} \quad m \in \text{meth}(\text{TA}) \\
\hline
\text{n} \notin \text{meth}(\text{TA}) \cup \text{meth}(\bar{S}) \quad \text{mtype}(m, \text{TA}) = \bar{I} \rightarrow \text{I} \\
\text{TA with } \{\text{m@n}\} \text{ OK requires } \bar{S} \cup \{\text{I m}(\bar{I} \bar{x})\} \quad (\text{Alias}\cdot\text{Ok}_1) \\
\\
\text{TA OK requires } \bar{S} \quad m \in \text{meth}(\text{TA}) \quad \text{n} \notin \text{meth}(\text{TA}) \\
\text{I n}(\bar{I} \bar{x}) \in \bar{S} \quad \text{mtype}(m, \text{TA}) = \bar{I} \rightarrow \text{I} \\
\hline
\text{TA with } \{\text{m@n}\} \text{ OK requires } (\bar{S} \setminus \{\text{I n}(\bar{I} \bar{x})\}) \cup \{\text{I m}(\bar{I} \bar{x})\} \quad (\text{Alias}\cdot\text{Ok}_2) \\
\\
\text{TA OK requires } \bar{S} \quad m \in \text{meth}(\text{TA}) \quad \text{mtype}(m, \text{TA}) = \bar{I} \rightarrow \text{I} \\
\hline
\text{TA minus } \{\text{m}\} \text{ OK requires } \bar{S} \cup \{\text{I m}(\bar{I} \bar{x})\} \quad (\text{Exclude}\cdot\text{Ok})
\end{array}$$

- (Alias·Ok₁) This rule handles the typechecking of aliasing where the new name n is not a required method. It checks that the new method name n does not correspond to a defined method in TA , and that the method name being aliased m exists. Then, simply adds the method m to the required methods.
- (Alias·Ok₂) Behaves as (Alias·Ok₁), except that the aliased method name m takes the place of a required method and that the new method name n must be removed from the list (both must have the same type interface).
- (Exclude·Ok) Adds the aliased method m to the required methods, in case another method calls m .

The class and trait typing rules are defined as follows

$$\begin{array}{c}
\cap \bar{\text{TA}} \setminus \diamond \bar{\text{TA}} \subseteq \text{meth}(\bar{\text{M}}) \quad \bar{\text{M}} \text{ OK IN T} \\
\hline
\bar{\text{TA}} \text{ OK requires } \bar{S}' \quad (\text{sig}(\bar{\text{TA}}) \cup \bar{S}) \text{ OK IN T} \\
\text{trait T imports } \bar{\text{TA}} \{\bar{\text{M}}; \bar{S}\} \text{ OK requires } (\bar{S} \cup \bar{S}') \setminus \text{meth}(\text{T}) \quad (\text{Tr}\cdot\text{Ok}) \\
\\
\text{K} = \text{C}(\bar{\text{J}} \bar{g}, \bar{\text{I}} \bar{f})\{\text{super}(\bar{g}); \text{this}.\bar{f} = \bar{f};\} \\
\text{fields}(\text{D}) = \bar{\text{J}} \bar{g} \quad \cap \bar{\text{TA}} \setminus \diamond \bar{\text{TA}} \subseteq \text{meth}(\bar{\text{M}}) \quad \text{sig}(\bar{\text{TA}}) \text{ OK IN C} \\
\hline
\bar{\text{M}} \text{ OK IN C} \quad \bar{\text{TA}} \text{ OK requires } \bar{S}' \quad \text{meth}(\bar{S}) \subseteq \text{meth}(\text{C}) \quad (\text{Cla}\cdot\text{Ok}) \\
\text{class C extends D imports } \bar{\text{TA}} \{\bar{\text{I}} \bar{f}; \text{K } \bar{\text{M}}\} \text{ OK}
\end{array}$$

Intuitively, those two rules check that all the components of the class and of the trait are well-typed, and that all conflicts are resolved; the trait rule also builds the list of methods that are required but not provided in the trait. In those rules, for $\text{TT}(\text{T}) = \text{trait T imports } \bar{\text{TA}} \{...\}$ we have a judgment $\bar{S} \text{ OK IN T}$ basically meaning that the methods whose signature are in \bar{S} do not raise a typing conflict with T . It is used to check that the imported traits are compatible (a class cannot import a trait with a method m returning an integer and another trait with a method m returning a string, for instance).

(Tr·Ok)/(Cla·Ok)

- (only in (Cla·Ok)) We fetch the constructor K and the fields \bar{g} .
- We typecheck the set of altered traits $\bar{\text{TA}}$ producing a set of required methods (the $\text{requires } \bar{S}'$ part).

- We typecheck the methods \bar{M} inside \mathbb{T}/\mathbb{C} .
- We check the key condition $\cap\bar{\mathbb{T}}\bar{\mathbb{A}} \setminus \diamond\bar{\mathbb{T}}\bar{\mathbb{A}} \subseteq \text{meth}(\bar{M})$ ensuring that every conflict is resolved, and guaranteeing that the lookup algorithm provides the correct conflict resolution.
- We typecheck the method signatures in $\bar{\mathbb{T}}\bar{\mathbb{A}}$ and \bar{S} . We check that all the traits $\bar{\mathbb{T}}\bar{\mathbb{A}}$ and the signature \bar{S} are compatible in the trait \mathbb{T} (resp. $\bar{\mathbb{T}}\bar{\mathbb{A}}$ in the class \mathbb{C}), *i.e.* they can be pairwise composed without any conflict in the method types (for instance, type conflicts may arise between an existing and an abstract method).
- (only in (Tr·Ok)) The set of required methods in \mathbb{T} , is then the set \bar{S}' of all methods signature required by the imported trait alterations $\bar{\mathbb{T}}\bar{\mathbb{A}}$ plus the set \bar{S} declared in \mathbb{T} except all the methods that are defined (*i.e.* implemented) in \mathbb{T} . It is worth noticing here that the user can define more method signatures in \bar{S} than what is really needed; in other words: if a method has been declared in \bar{S} but its behavior is already present inside an imported trait, then the system only checks that both the type of the defined method and of the required one are compatible.
- (only in (Cla·Ok)) We check the condition $\text{meth}(\bar{S}) \subseteq \text{meth}(\mathbb{C})$. It means that classes have to provide all the necessary methods for the computation of its instances (objects of the form `new C(...)`). Specifically, $\text{meth}(\bar{S})$ are being implemented either inside the class, in the superclass or inside some trait (other than the ones requiring it). As such, we ensure that every time a method m is expected for \mathbb{C} then it is also implemented in \mathbb{C} .

4 Properties

Once the type system for iFTJ has been set up, the next step is to prove that (i) the static semantics matches the dynamic one, *i.e.*, types are preserved during computation (modulo subtyping), that (ii) the interpreter cannot get stuck if programs only include upcasts, and finally that (iii) the type system prevents programs from the run-time *message-not-understood* error. The proofs of these statements are not excessively more complicated than in FJ. The full proofs are provided in Appendix B.

Untypable programs are not necessarily deterministic, since the lookup rules do not give the priority to any of the imported trait (so that the order in which traits are importer does not change the semantics). Conflict Resolution Theorem states that typed programs have a deterministic lookup algorithm. Which means nothing more than saying that all conflicts have been resolved.

Theorem 1 (Conflict Resolution)

If, for all $\mathbb{C}_i \in \text{CL}$, we have $\mathbb{C}_i \text{ OK}$, then both $m\text{body}$ and $m\text{type}$ are functions. \square

Subject reduction follows easily.

Theorem 2 (Subject Reduction)

If $\Gamma \vdash e \in C$ and $e \longrightarrow e'$, then $\Gamma \vdash e' \in D$, for some $D <: C$. \square

Progress shows that the only way for the interpreter to get stuck is by reaching a state where a downcast is impossible. Let $\#$ means cardinality (as in [9]).

Theorem 3 (Progress)

Suppose e is a well-typed expression

- (1) If e includes $\text{new } C(\bar{e}).f$ as a subexpression, then $\text{fields}(C) = \bar{T} \bar{f}$ and $f \in \bar{f}$;
- (2) If e includes $\text{new } C(\bar{e}).m(\bar{f})$ as a subexpression, then $\text{mbody}(m, C) = (\bar{x}, e_0)$ and $\#(\bar{x}) = \#(\bar{d})$. \square

In accordance with FJ, we define the notion of *safe* expression e in Γ if the type derivation of the underlying (CT, TT) and $\Gamma \vdash e \in C$ contains no downcast or *stupid cast* (rules (Typ·DCast), and (Typ·SCast)). Then, soundness of safety and progress of safe programs follow.

Theorem 4 (Reduction preserves safety)

If e is safe in Γ , and $e \longrightarrow e'$, then e' is safe in Γ . \square

Theorem 5 (Progress of safe programs)

Suppose e is safe in Γ . If e has $(C)\text{new } D(\bar{e})$ as a subexpression, then $D <: C$. \square

5 Example

We give a small example in Figure 2, using the Java class `Integer` enriched with some simple algebraic methods, *e.g.* `mod` and `times`. Here is a sum-up of this example. First we define a trait `Convertible` which is purely abstract (as an interface in Java, we define it only for typing purposes); it requires a method producing an integer. Then, we declare a trait `Hashable` that imports `Convertible`, and uses the `to_Int` method as an input for its hashing function. It allows then to define an extension `H_Integer` of the class `Integer` with a method `hash` to get a hash value of the considered integer. Independently, we define a trait `Convertible_Pair` that imports `Convertible` (thus every `Convertible_Pair` is also `Convertible`) and define a method `to_Int` for a pair of two convertible object. Finally, we define our strings as lists of characters. The strings are indeed subclasses of a trait `My_String`, which is an interface equivalent to `Hashable` (in real life it would be a strict subtype of `Hashable` though). As such, we have two classes that build strings, namely `Null_String`, *i.e.* a single element class, and `Cons_String` which is intrinsically a pair of a character and a string. The class `Cons_String` imports `Convertible_Pair` to implement the method `to_Int` which is required by `My_String` (as a trait importing `Hashable`). Then, we can assume a class `Array`, which can be used as an array or an association table. Together with the hashing function of strings (or that of hashable integers) it may be used as a hash table, as suggested

```

trait Convertible {;Integer to_Int()}
trait Hashable imports Convertible {
  Integer hash(){... this.to_Int() ...};}
trait Convertible_Pair imports Convertible {
  Integer to_Int(){return this.fst().to_Int().
    plus(this.snd().to_Int().times(this.offset()));};
  Convertible fst()
  Convertible snd()
  Integer offset()}
trait My_String imports Hashable { ; }

class H_Integer extends Integer imports Hashable {;
  H_Integer() {super();}
  Integer to_Int() {return this;} }
class My_Char extends Objects imports Convertible {
  int me;
  My_Char(Integer c) {super();this.me=c.mod(new Integer(256));}
  Integer to_Int() {return this.me;} }
class Cons_String extends Objects imports My_String Convertible_Pair {
  My_Char head
  My_String tail;
  Cons_String(My_Char c,My_String s)
    {super();this.head=c; this.tail=s;}
  Convertible fst() {return this.head;}
  Convertible snd() {return this.tail;}
  Integer offset() {return new Integer(256);} }
class Null_String extends Objects imports My_String {;
  Null_String() {super();}
  Integer to_Int() {return new Integer(0);} }
class Array extends Objects imports H_Integers {...
  Object index(Integer i) {...}
  Object assoc(Integer i) {...} }

(Array)(new Array(...).index(new Cons_String(...).hash()))
    .assoc(new Cons_String(...))

```

Figure 2. Hashing Strings

in the example. Moreover, `H_Integers`, `Cons_String`, and `Null_String` do not only share the `Hashable` type, but they also have a common implementation of the method `hash` gotten through trait inheritance.

6 Related Work and Conclusions

Related Work. In the past few years, great interest was recorded around the possibility to use trait inheritance in statically typed class- and object-based languages [10, 13, 14, 18]. Among the many propositions which arose in the literature (and apart our FTJ), we recall the following ones.

- (**TcoreMoby**) adds statically typed trait inheritance to an object-based calculus with first-class functions of the ML family. Fisher and Reppy have the same interest in typed traits as we do, and historically this paper can be considered as the first attempt to typecheck traits statically. The key points of **TcoreMoby** are that (a) two traits can be combined only if they are disjoint, and that (b) one method can be overridden by another only if it has the same type interface, and that (c) in **TcoreMoby** traits need to be typechecked only once. The paper comes with the full set of proofs. Our **iFTJ** relaxes point (a) and features points (b) and (c).
- (**Chai**) adds statically typed trait inheritance to a Java-like language; in fact there are three dialects defined: **Chai**_{1,2,3}. As for **TcoreMoby**, the key points in **Chai** are that (a) two traits can be combined only if they are disjoint, (b) one method can be overridden by another only if it has the same type interface, and (c) in **Chai**_{2,3} traits are typechecked only once, and that (d) in **Chai**₃ traits can be substituted for one another dynamically. The paper comes with proof sketches for the theorems of **Chai**₁, and soundness theorems for **Chai**_{2,3}, whose proofs are not yet published. Our **iFTJ** can be compared with **Chai**₂: the bigger difference is that **iFTJ** relaxes point (a), by allowing conflicts to be resolved via overriding, and features points (b) and (c), making the type system more expressive than **Chai**₂. Moreover, **iFTJ** comes with a full metatheory.
- (**Scala**) features traits as specific instance of an abstract class; thus the abstract modifier is redundant for it. Traits in **Scala** are a bit like interfaces in **ClassicJava** [8], since they are used to define object types by specifying the signature of the supported methods. Besides in **Scala** the composition order of trait is irrelevant. A solid implementation is available on the **Scala** web site. A **Featherweight Scala** formal model with related meta-theory remains to be fleshed out (and a formal comparison of features also).
- (**Fortress**) specification language by Allen, Chase, Luchangco, Maessen, Ryu, Steele, and Tobin-Hochstadt was published on **SUN**'s website at the end of 2005. This language features traits-as-types (*i.e.* a trait is like an interface in Java with some concrete method bodies inside), and objects are trait instances, obtained by completing the imported trait by the body declaration of the abstract methods. A formal model with related meta-theory remains to be fleshed out.

However, our type system is deeply indebted to the work on *incomplete objects* by Bono, Bugliesi, Dezani, and Liquori [3]; this work presented a type system for the *Lambda Calculus of Objects* of Fisher, Honsell, and Mitchell [6], an untyped λ -calculus enriched with object primitives. The paper allowed objects to be typed independently of the order of their method additions. This flexibility arises from introducing the notion of *completion*, a complement to *interface*, to convey information on (the types of) the methods which are not available in the object, and yet are referenced by its the methods. Besides allowing a more flexible typing of methods (in particular, of mutually

recursive method definitions), this extension also allows methods to be invoked on *incomplete* objects, *i.e.* objects whose implementation (the set of their methods) is only partially specified. The paper conjectured that the concept underpinning the typing of incomplete objects may be exploited in modeling language constructs such as virtual methods and interfaces in class-based languages (exactly what our model of typed trait does). Ten year later, we found some evidence in our conjecture in designing a type system for iFTJ.

Conclusions. We have presented a formal development of iFTJ, a statically typed, functional, class-based language featuring classes, objects, trait inheritance where traits need to be typechecked only once in order to make the system compatible with separated compilation. Future directions will focus on:

- Add bounded polymorphic-types or even generic-types as in GJ [9]; this extension will greatly improve the usefulness of statically typed traits.
- Study the impact of trait inheritance for the language C#; although this language is quite similar to Java, it has its peculiarities, which should be carefully interleaved and kept compatible with typed traits.

Acknowledgments. (By the first author, in italian.) *Cari Mario, Mariangiola e Simona, senza i vostri insegnamenti non saprei quello che so e non sarei quello che sono ...*

References

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
- [2] E. Allen, D. Chase, V. Luchangco, J-W Maessen, G.L.Steele S. Ryu, and S. Tobin-Hochstadt. The Fortress Language Specification, version 0.618., 2005. <http://research.sun.com/projects/plrg/fortress0618.pdf>.
- [3] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping for Extensible, Incomplete Objects. *Fundamenta Informaticae*, 38(4):325–364, 1999.
- [4] L. Cardelli. Obliq: A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
- [5] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.
- [6] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.
- [7] K. Fisher and J. Reppy. Statically Typed Traits. http://www.cs.uchicago.edu/files/tr_authentic/TR-2003-13.pdf. The early version “A Typed Calculus of Traits” has been presented at FOOL 10, 2004.

- [8] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. of POPL*, pages 171–183. The ACM Press, 1998.
- [9] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
- [10] L. Liquori and A. Spiwack. *FeatherTrait: a modest extension of Featherweight Java*. *ACM TOPLAS*, to appear, 200X.
- [11] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [12] The Moby Team. The Moby Home Page, <http://moby.cs.uchicago.edu/>.
- [13] O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening Traits. *Journal of Object Technology*, 5(3), 2006.
- [14] P. J. Quitslung. Java Traits – Improving Opportunities for Reuse. Technical Report CSE-04-005, OGI School of Science and Engineering, 2004.
- [15] The Scala Team. The Scala Home Page, 2007. <http://scala.epfl.ch/>.
- [16] N. Schärli. *Traits - Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Berne, 2005.
- [17] N. Schärli, S. Ducasse, O. Nierstrasz, and A.P. Black. Traits: Composable Units of Behaviour. In *Proc. of ECOOP*, volume 2743 of *LNCS*, pages 248–274. Springer-Verlag, 2003.
- [18] C. Smith and S. Drossopoulou. Chai: Typed Traits in Java. In *Proc. of ECOOP*, volume 3586 of *LNCS*, pages 453–478. Springer Verlag, 2005.
- [19] A. Snyder. Inheritance and the Development of Encapsulated Software Systems. In *Research Directions in Object-Oriented Programming*, pages 165–188. MIT Press, 1987.
- [20] D. Ungar and B. Smith, R. Self: The Power of Simplicity. In *Proc. of OOPSLA*, pages 227–241. The ACM Press, 1987.

A Dynamic and Static Semantics of iFTJ

Head and Subtyping (Sub·Tr) and (Sub·Cla·Tr) plus

$$head(T) = T \quad head(TA \text{ with } \{m@n\}) = head(TA) \quad head(TA \text{ minus } \{m\}) = head(TA)$$

$$\frac{}{I <: I} \text{(Sub·Refl)} \quad \frac{I_1 <: I_2 \quad I_2 <: I_3}{I_1 <: I_3} \text{(Sub·Trans)} \quad \frac{\text{class } C \text{ extends } D \text{ imports } \overline{TA} \{ \dots \}}{C <: D} \text{(Sub·Cla)}$$

Small-step semantics

$$\frac{fields(C) = \overline{I} \overline{f}}{(\text{new } C(\overline{e})) \cdot f_i \longrightarrow e_i} \text{(Run·Field)} \quad \frac{C <: I}{(I)(\text{new } C(\overline{e})) \longrightarrow \text{new } C(\overline{e})} \text{(Run·Cast)}$$

$$\frac{mbody(m, C) = (\overline{x}, e_0)}{(\text{new } C(\overline{e})) \cdot m(\overline{d}) \longrightarrow [\overline{d}/\overline{x}, \text{new } C(\overline{e})/\text{this}]e_0} \text{(Run·Call)}$$

Congruence

$$\frac{e \longrightarrow e'}{e \cdot f \longrightarrow e' \cdot f} \text{(Cgr·Field)} \quad \frac{e \longrightarrow e'}{e \cdot m(\overline{e}) \longrightarrow e' \cdot m(\overline{e})} \text{(Cgr·Receiver)} \quad \frac{e \longrightarrow e'}{(I)e \longrightarrow (I)e'} \text{(Cgr·Cast)}$$

$$\frac{e_i \longrightarrow e'_i}{e \cdot m(\dots, e_i, \dots) \longrightarrow e \cdot m(\dots, e'_i, \dots)} \text{(Cgr·Args)} \quad \frac{e_i \longrightarrow e'_i}{\text{new } C(\dots, e_i, \dots) \longrightarrow \text{new } C(\dots, e'_i, \dots)} \text{(Cgr·New)}$$

Field lookup exactly as in FJ

Method body lookup (MBdy·Tr) and (MBdy·SCla) plus

$$CT(C) = \text{class } C \text{ extends } D \text{ imports } \overline{TA} \{ \overline{I} \overline{f}; K \overline{M} \}$$

$$\frac{I \ m(\overline{I} \ \overline{x}) \{ \text{return } e; \} \in \overline{M}}{mbody(m, C) = (\overline{x}, e)} \text{(MBdy·Cla)}$$

Trait lookup

$$\frac{\exists TA \in \overline{TA}. \ altlook(m, TA) = M}{tlook(m, \overline{TA}) = M} \text{(Tr·Ok)} \quad \frac{\forall TA \in \overline{TA}. \ altlook(m, TA) = fail}{tlook(m, \overline{TA}) = fail} \text{(Tr·Ko)}$$

Trait alteration lookup (ATr·Inh) and (ATr·Ali₁) plus

$$\frac{TT(T) = \text{trait } T \text{ imports } \overline{TA} \{ \overline{M}; \overline{S} \} \quad I \ m(\overline{I} \ \overline{x}) \{ \text{return } e; \} \in \overline{M}}{altlook(m, T) = I \ m(\overline{I} \ \overline{x}) \{ \text{return } e; \}} \text{(ATr·Found)}$$

$$\frac{m \neq p \quad m \neq q \quad altlook(m, TA) = M_{\perp}}{altlook(m, TA \text{ with } \{p@q\}) = M_{\perp}} \text{(ATr·Ali}_2\text{)}$$

$$\frac{m \neq n}{altlook(m, TA \text{ with } \{m @ n\}) = fail} \text{(ATr·Ali}_3\text{)} \quad \frac{altlook(n, TA) = fail}{altlook(m, TA \text{ with } \{n @ m\}) = fail} \text{(ATr·Ali}_4\text{)}$$

$$\frac{m \neq n \quad altlook(m, TA) = M_{\perp}}{altlook(m, TA \text{ minus } \{n\}) = M_{\perp}} \text{(ATr·Exl}_1\text{)} \quad \frac{}{altlook(m, TA \text{ minus } \{m\}) = fail} \text{(ATr·Exl}_2\text{)}$$

Method names and signatures

$$\frac{}{meth(I \ m(\bar{I} \ \bar{x})) = \{m\}} \text{(Mth·Sig)}$$

$$\frac{}{meth(I \ m(\bar{I} \ \bar{x})\{return \ e; \}) = \{m\}} \text{(Mth·Mth)}$$

$$\frac{TT(T) = \text{trait } T \text{ imports } \bar{T}A \ \{\bar{M}; \bar{S}\}}{meth(T) = meth(\bar{M}) \cup meth(\bar{T}A)} \text{(Mth·Tr)}$$

$$\frac{CT(C) = \text{class } C \text{ extends } D \text{ imports } \bar{T}A \ \{\bar{C} \ \bar{f}; K \ \bar{M}\}}{meth(C) = meth(\bar{M}) \cup meth(\bar{T}A) \cup meth(D)} \text{(Mth·Cla)}$$

$$\frac{}{meth(TA \text{ with } \{m @ n\}) = (meth(TA) \setminus \{m\}) \cup \{n\}} \text{(Mth·Ali)}$$

$$\frac{}{meth(TA \text{ minus } \{m\}) = meth(TA) \setminus \{m\}} \text{(Mth·Exl)}$$

$$\frac{}{sig(I \ m(\bar{I} \ \bar{x})\{\dots\}) = \{I \ m(\bar{I} \ \bar{x})\}} \text{(Sig·Mth)}$$

$$\frac{sig(TA) = \{\bar{S}, I \ m(\bar{I} \ \bar{x})\}}{sig(TA \text{ with } \{m @ n\}) = \{\bar{S}, I \ m(\bar{I} \ \bar{x}), I \ n(\bar{I} \ \bar{x})\}} \text{(Sig·Alias)}$$

$$\frac{sig(TA) = \{\bar{S}, I \ m(\bar{I} \ \bar{x}); \}}{sig(TA \text{ minus } \{m\}) = \{\bar{S}\}} \text{(Sig·Exl)} \quad \frac{TT(T) = \text{trait } T \text{ imports } \bar{T}A \ \{\bar{M}; \bar{S}\}}{sig(T) = \bar{S} \cup sig(\bar{M}) \cup sig(\bar{T}A)} \text{(Sig·Tr)}$$

Method paths in trait alterations (Path·{Inh, Alias_{1,2}, Exc}) plus

$$\frac{m \in meth(TA)}{m \text{ in } TA \trianglelefteq m \text{ in } TA} \text{(Path·Refl)} \quad \frac{m \text{ in } TA_1 \trianglelefteq p \text{ in } TA_2 \quad p \text{ in } TA_2 \trianglelefteq n \text{ in } TA_3}{m \text{ in } TA_1 \trianglelefteq n \text{ in } TA_3} \text{(Path·Trans)}$$

Method type lookup

$$\frac{CT(C) = \text{class } C \text{ extends } D \text{ imports } \bar{T}A \ \{\bar{J} \ \bar{f}; K \ \bar{M}\} \quad I \ m(\bar{I} \ \bar{x})\{return \ e; \} \in \bar{M}}{mtype(m, C) = \bar{I} \rightarrow I} \text{(MTyp·Self)}$$

$$\frac{\text{altlook}(m, \text{TA}) = \text{I } m(\bar{\text{I}} \bar{x}) \{\text{return } e; \}}{m\text{type}(m, \text{TA}) = \bar{\text{I}} \rightarrow \text{I}} \text{(MTyp·Impl)}$$

$$\frac{\text{altlook}(m, \text{TA}) = \text{fail} \quad m\text{sig}(m, \text{TA}) = \text{fail}}{m\text{type}(m, \text{TA}) = \text{fail}} \text{(MTyp·Fail)}$$

Signature lookup and overriding (MSig·{Inh, End, Ali₃}) plus

$$\frac{\text{TT}(\text{T}) = \text{trait } \text{T imports } \bar{\text{TA}} \{\bar{\text{M}}; \bar{\text{S}}\} \quad m \notin \text{meth}(\text{T}) \quad \text{I } m(\bar{\text{I}} \bar{x}) \in \bar{\text{S}}}{m\text{sig}(m, \text{T}) = \text{I } m(\bar{\text{I}} \bar{x})} \text{(MSig·Tr)}$$

$$\frac{\text{TT}(\text{T}) = \text{trait } \text{T imports } \bar{\text{TA}} \{\bar{\text{M}}; \bar{\text{S}}\} \quad m \notin \text{meth}(\text{T}) \\ \forall \text{TA} \in \bar{\text{TA}}. m\text{sig}(m, \text{TA}) = \text{fail} \quad m \notin \text{meth}(\bar{\text{S}})}{m\text{sig}(m, \text{T}) = \text{fail}} \text{(MSig·End)}$$

$$\frac{m \neq p \quad m \neq q \quad m\text{sig}(m, \text{TA}) = \text{S}_\perp}{m\text{sig}(m, \text{TA with } \{p@q\}) = \text{S}_\perp} \text{(MSig·Ali}_1\text{)}$$

$$\frac{m \neq n}{m\text{sig}(m, \text{TA with } \{n@m\}) = \text{fail}} \text{(MSig·Ali}_2\text{)} \quad \frac{m \neq n \quad \text{altlook}(m, \text{TA}) = \text{fail}}{m\text{sig}(m, \text{TA with } \{m@n\}) = \text{fail}} \text{(MSig·Ali}_4\text{)}$$

$$\frac{\text{altlook}(m, \text{TA}) = \text{I } m(\bar{\text{I}} \bar{x}) \{\text{return } e; \}}{m\text{sig}(m, \text{TA minus } \{m\}) = \text{I } m(\bar{\text{I}} \bar{x})} \text{(MSig·Ex}_1\text{)} \quad \frac{\text{altlook}(m, \text{TA}) = \text{fail}}{m\text{sig}(m, \text{TA minus } \{m\}) = \text{fail}} \text{(MSig·Ex}_2\text{)}$$

$$\frac{m \neq n \quad m\text{sig}(m, \text{TA}) = \text{S}_\perp}{m\text{sig}(m, \text{TA minus } \{n\}) = \text{S}_\perp} \text{(MSig·Ex}_3\text{)} \quad \frac{m \in \text{meth}(\text{T})}{m\text{sig}(m, \text{T}) = \text{fail}} \text{(MSig·Fail)}$$

Valid Method Overriding

$$\frac{m\text{type}(m, \text{I}) = \bar{\text{J}} \rightarrow \text{J}_0 \text{ implies } \bar{\text{I}} = \bar{\text{J}} \text{ and } \text{I}_0 = \text{J}_0}{\text{override}(m, \text{I}, \bar{\text{I}} \rightarrow \text{I}_0)} \text{(M·Ov)}$$

Basic expression typing exactly as in FJ

Method typing (Tr·Ok) and (Cla·Ok) plus

$$\frac{\text{TT}(\text{T}) = \text{trait } \text{T imports } \bar{\text{TA}} \{\bar{\text{M}}; \bar{\text{S}}\} \quad \text{override}(m, \bar{\text{TA}}, \bar{\text{I}} \rightarrow \text{I})}{\text{I } m(\bar{\text{I}} \bar{x}) \text{ OK IN T}} \text{(Sig·Ok·Tr)}$$

$$\frac{\text{CT}(\text{C}) = \text{class } \text{C extends } \text{D imports } \bar{\text{TA}} \{\bar{\text{I}} \bar{f}; \text{K } \bar{\text{M}}\} \\ \text{override}(m, \text{D}, \bar{\text{I}} \rightarrow \text{I}) \quad \text{override}(m, \bar{\text{TA}}, \bar{\text{I}} \rightarrow \text{I})}{\text{I } m(\bar{\text{I}} \bar{x}) \text{ OK IN C}} \text{(Sig·Ok·Cla)}$$

B The Full Proofs

We prove that a method path relation only designs paths for existing methods.

Lemma 1 (Non Virtual Paths)

If m in $\text{TA}_1 \triangleleft n$ in TA_2 then, $m \in \text{meth}(\text{TA}_1)$ and $n \in \text{meth}(\text{TA}_2)$.

Proof By induction on the derivation of m in $\text{TA}_1 \triangleleft n$ in TA_2 . \square

We show that *altlook* provides a method implementation with the proper name.

Lemma 2 (Naming Soundness)

- If $\text{altlook}(m, \text{TA}) = M_\perp$, then either $M_\perp = \text{fail}$, or $M_\perp = \text{I } m (\bar{\text{I}} \bar{x}) \{ \dots \}$.
- If $\text{msig}(m, \text{TA}) = S_\perp$, then either $S_\perp = \text{fail}$, or $S_\perp = \text{I } m (\bar{\text{I}} \bar{x})$.

Proof

- Straightforward induction on the derivation of $\text{altlook}(m, \text{TA}) = M_\perp$.
- Follows straightforwardly from the first point. \square

We prove that a method path relation preserves the body of the method. It is the first step for proving determinism of well-typed programs.

Lemma 3 (Diamond Proto-Soundness)

If m in $\text{TA}_1 \triangleleft n$ in TA_2 , then $\text{altlook}(n, \text{TA}_2) = \text{I } n (\bar{\text{I}} \bar{x}) \{ \text{return } e; \}$ implies $\text{altlook}(m, \text{TA}_1) = \text{I } m (\bar{\text{I}} \bar{x}) \{ \text{return } e; \}$.

Proof By induction on the derivation of m in $\text{TA}_1 \triangleleft n$ in TA_2 . Here are the most relevant cases

- (Path·Inh) Since $m \notin \text{meth}(\bar{M})$, the rule (ATr·Inh) can apply to TA_1 which implies the result.
- (Path·Ali₁) The rule (ATr·Ali₁) (or (ATr·Ali₄)) can apply to TA_1 which implies the result.
- (Path·Ali₂) Since $m \neq p$ and $m \neq q$, the rule (ATr·Ali₂) can apply to TA_1 which implies the result.
- (Path·Exl) Since $m \neq p$, the rule (ATr·Exl₁) can apply to TA_1 which implies the result. \square

We prove that if a trait is well-typed, then *meth* refers to the set of methods where *altlook* does not fail.

Lemma 4 (*meth* Soundness)

If TA OK requires \bar{S} , then $m \in \text{meth}(\text{TA})$ if and only if $\text{altlook}(m, \text{TA}) \neq \text{fail}$.

Proof By induction on the derivation of $\text{altlook}(m, \text{TA})$. Here are the most relevant cases

- (ATr·Found) Then, $\text{TA} = \text{T}$. Then, $\text{altlook}(m, \text{T}) \neq \text{fail}$ and, from rule (Mth·Tr), we have $m \in \text{meth}(\text{T})$.
- (ATr·Inh) Then, $\text{TA} = \text{T}$, and $\text{TT}(\text{T}) = \text{trait } \text{T imports } \bar{\text{TA}} \{ \bar{M}; \bar{S} \}$. Then,

$\text{altlook}(m, T) \neq \text{fail} \iff \exists TA_i \in \overline{TA}. \text{altlook}(m, TA_i) \neq \text{fail}$. Thus we have, by induction hypothesis

$\text{altlook}(m, T) \neq \text{fail} \iff \exists TA_i \in \overline{TA}. m \in \text{meth}(m, TA_i) \iff m \in \text{meth}(T)$.

The latter comes from rule (Mth·Tr) and the statement $m \notin \text{meth}(\overline{M})$.

- (ATr·Ali₁) Then, $TA = TA_1$ with $\{n@m\}$. Since TA is well-typed, we have that $n \in \text{meth}(TA_1)$ and $m \notin \text{meth}(TA_1)$. Then, by induction hypothesis, we have that $\text{altlook}(n, TA_1) \neq \text{fail}$, thus $\text{altlook}(m, TA) \neq \text{fail}$, and rule (Mth·Ali) states that $m \in \text{meth}(TA)$. \square

We prove that altlook is a function when the program typechecks.

Lemma 5 (Conflict Resolution in Trait Alterations)

If TA OK requires \overline{S} , then $\text{altlook}(\cdot, TA)$ is a function.

Proof By induction on the derivation of altlook . Here are the most relevant cases

- (ATr·Inh) If $\text{tlook}(m, \overline{TA}) = \text{fail}$, then the property obviously holds. Else, by induction hypothesis, for all $TA_i \in \overline{TA}$, $\text{altlook}(\cdot, TA_i)$ is a function.
 - If $m \notin \cap \overline{TA}$, then there is an unique $TA_i \in \overline{TA}$ where $\text{altlook}(m, TA_i) \neq \text{fail}$.
 - If $m \in \cap \overline{TA}$, then, since TA is well-typed, the rule (Tr·Ok) enforces that $m \in \diamond \overline{TA}$. Then, for all $TA_i \in \overline{TA}$, we have $m \in \text{meth}(TA_i) \Rightarrow m \text{ in } TA_i \triangleleft n \text{ in } TA_1$. Moreover, we know that $n \in \text{meth}(TA_1)$, by Lemma 1, which means that there is at least one $\text{altlook}(n, TA_1) = I n (\overline{I} x)\{\dots\}$ which is derivable, by Lemma 4. Thus, $\text{altlook}(m, TA_i) = I m (\overline{I} x)\{\dots\}$ is derivable for all TA_i such that $m \in \text{meth}(TA_i)$, by Lemma 3. To conclude, we know that $\text{altlook}(\cdot, TA_i)$ is a function which ensures they are all equal.
- (ATr·Ali₁) $TA = TA_1$ with $\{n@m\}$. The induction hypothesis ensures that $\text{altlook}(\cdot, TA_1)$ is a function. Then, it is straightforward. \square

We prove that sig returns the set of the signatures of all method (both implemented and required) of a typed trait.

Lemma 6 (sig Soundness)

If TA OK requires \overline{S} , then $\text{mtype}(m, TA) = \overline{I} \rightarrow I$ if and only if $I m (\overline{I} \overline{x}) \in \text{sig}(TA)$.

Proof We prove, first, that for any $m \in \text{meth}(TA)$, we have $\text{mtype}(m, TA) = \overline{I} \rightarrow I$ for some I, \overline{I} and for the same I, \overline{I} , we have $I m (\overline{I} \overline{x}) \in \text{sig}(TA)$. This reduces to the fact that $\text{altlook}(m, TA) = I m (\overline{I} \overline{x})\{\dots\}$ implies $I m (\overline{I} \overline{x}) \in \text{sig}(TA)$, by Lemma 4, and thanks to the rule (MTyp·Impl). We prove the latter result by straightforward induction on the derivation of $\text{altlook}(m, TA) = I m (\overline{I} \overline{x})\{\dots\}$. To conclude, we now need to establish the fact that if $m \notin \text{meth}(TA)$, then we have $\text{mtype}(m, TA) = \overline{I} \rightarrow I$ if and only if $I m (\overline{I} \overline{x}) \in \text{sig}(TA)$. This fact is equivalent to if $m \notin \text{meth}(TA)$, then $\text{msig}(m, TA) = I m (\overline{I} \overline{x})$ if and only if $I m (\overline{I} \overline{x}) \in \text{sig}(TA)$. Since it is obvious that we cannot derive both $\text{msig}(m, TA) = I m (\overline{I} \overline{x})$ and $\text{msig}(m, TA) = \text{fail}$, we can prove both directions separately (and that either of them holds). We prove that if $m \notin \text{meth}(TA)$ and $\text{msig}(m, TA) = I m (\overline{I} \overline{x})$, then $I m (\overline{I} \overline{x}) \in \text{sig}(TA)$, and that if $m \notin \text{meth}(TA)$ and $\text{msig}(m, TA) = \text{fail}$,

then $\mathbb{I} \ m(\bar{\mathbb{I}} \ \bar{x}) \notin \text{sig}(\text{TA})$. Both are proved by straightforward induction on the derivation of $\text{msig}(m, \text{TA}) = \mathbb{S}_\perp$. We will emphasize one case of each, both being representative of the proofs.

- (MSig·Ali₃) Since $\text{altlook}(m, \text{TA}) = \mathbb{I} \ m(\bar{\mathbb{I}} \ \bar{x})\{\dots\}$, then we know, thanks to the former part of the proof, that $\mathbb{I} \ m(\bar{\mathbb{I}} \ \bar{x}) \in \text{sig}(\text{TA})$. Then, by rule (Sig·Alias), we derive that $\mathbb{I} \ m(\bar{\mathbb{I}} \ \bar{x}) \in \text{sig}(\text{TA with } \{m@n\})$. Hence the result.
- (MSig·Ali₄) Thanks to typechecking of TA with $\{m@n\}$, we know that $m \in \text{meth}(\text{TA})$. By Lemma 4, we thus know that $\text{altlook}(m, \text{TA}) \neq \text{fail}$. This case can't raise. \square

We prove that when the program typechecks, msig is a function. It is necessary to ensure soundness of typing.

Lemma 7 (Conflict Resolution in Abstract Signatures)

If TA OK requires $\bar{\mathbb{S}}$, then $\text{msig}(\cdot, \text{TA})$ is a function (modulo renaming of the formal parameters).

Proof By induction on a derivation of msig . All cases are obviously disjoint, except for (MSig·Ex₁) / (MSig·Ex₂), and (MSig·Ali₃) / (MSig·Ali₄) which apply to the same terms. Remember however that by Lemma 5, $\text{altlook}(\cdot, \text{TA})$ is a function, thus ensuring that those two pairs of rules are indeed disjoint. Knowing this and the fact that $\text{altlook}(\cdot, \text{TA})$ is a function, every rule of msig is straightforward, except from (MSig·Inh) treated below.

- (MSig·Inh) Then, T has been typechecked through rule (Tr·Ok), and $\text{TT}(\text{T}) = \text{trait } \text{T imports } \bar{\text{TA}} \ \{\bar{\text{M}}; \bar{\text{S}}\}$. We have, incidentally, that all $\text{TA} \in \bar{\text{TA}}$ do typecheck also. Thus, by Lemma 5, $\text{altlook}(\cdot, \text{TA})$ is a function for all $\text{TA} \in \bar{\text{TA}}$ and, by induction hypothesis, $\text{msig}(\cdot, \text{TA})$ is a function for all $\text{TA} \in \bar{\text{TA}}$. We also have that $m \notin \text{meth}(\bar{\text{TA}})$, by Lemma 4, and this means that $\text{altlook}(m, \text{TA}) = \text{fail}$ for all $\text{TA} \in \bar{\text{TA}}$ (and it can be nothing else since $\text{altlook}(\cdot, \text{TA})$ is a function). We can then conclude that $\text{mtype}(m, \text{TA}_i)$ is inferred through the (MTyp·Virt) rule for each $\text{TA}_i \in \bar{\text{TA}}$. Now let's assume that there are $\text{TA}_1, \text{TA}_2 \in \bar{\text{TA}}$ such that $\text{msig}(m, \text{TA}_1) \neq \text{fail}$ and $\text{msig}(m, \text{TA}_2) \neq \text{fail}$. Then, $\text{msig}(m, \text{TA}_1) = \mathbb{I} \ m(\bar{\mathbb{I}} \ \bar{x})$ and $\text{msig}(m, \text{TA}_2) = \mathbb{J} \ m(\bar{\mathbb{J}} \ \bar{y})$, and we can then deduce that $\text{mtype}(m, \text{TA}_1) = \bar{\mathbb{I}} \rightarrow \mathbb{I}$ and $\text{mtype}(m, \text{TA}_2) = \bar{\mathbb{J}} \rightarrow \mathbb{J}$ are derivable. The judgment $(\text{sig}(\bar{\text{TA}}) \cup \bar{\mathbb{S}}) \text{ OK IN T}$ in (Tr·Ok), ensures then that $\bar{\mathbb{I}} = \bar{\mathbb{J}}$ and $\mathbb{I} = \mathbb{J}$, by Lemma 6. Hence the result. \square

The system is kept non-deterministic to emphasize the fact that the order of trait composition does not matter in the result. We prove that all conflict are resolved both for static (typing) and dynamic semantics.

Theorem 1 (Conflict Resolution)

If for all $\text{C}_i \in \text{CL}$, we have $\text{C}_i \text{ OK}$, and for all $\text{T}_i \in \text{TT}$, we have $\text{T}_i \text{ OK requires } \bar{\mathbb{S}}$, then both $\text{mbody}(\cdot, \cdot)$ and $\text{mtype}(\cdot, \cdot)$ are functions.

Proof We prove that $\text{mbody}(\cdot, \cdot)$ is a function by induction on the derivation of $\text{mbody}(m, \text{C}_i)$, the proof for $\text{mtype}(\cdot, \cdot)$ being similar (note however that

there are two extra cases to deal with for *mtype*, handled directly by Lemmas 5 and 7).

- (MBdy·Cla) *Direct.*
- (MBdy·SCla) *Straightforward by induction hypothesis.*
- (MBdy·Tr) *For all $\text{TA}_i \in \overline{\text{TA}}$, $\text{altlook}(\cdot, \text{TA}_i)$ is a function by Lemma 5.*
 - *If $\mathfrak{m} \notin \cap \overline{\text{TA}}$, then there is an unique $\text{TA}_i \in \overline{\text{TA}}$ where $\text{altlook}(\mathfrak{m}, \text{TA}_i) \neq \text{fail}$.*
 - *If $\mathfrak{m} \in \cap \overline{\text{TA}}$, then, since \mathcal{C}_i is well-typed, the rule (Cla·Ok) enforces that $\mathfrak{m} \in \diamond \overline{\text{TA}}$. Then, for all $\text{TA}_i \in \overline{\text{TA}}$, we have $\mathfrak{m} \in \text{meth}(\text{TA}_i) \Rightarrow \mathfrak{m} \text{ in } \text{TA}_i \triangleleft \mathfrak{n} \text{ in } \text{TA}_i$. Moreover, we know that $\mathfrak{n} \in \text{meth}(\text{TA}_i)$, by Lemma 1, which means that there is at least one $\text{altlook}(\mathfrak{n}, \text{TA}_i) = \text{I } \mathfrak{n} (\overline{\text{I}} \text{ x}) \{ \dots \}$ which is derivable, by Lemma 4. Thus, $\text{altlook}(\mathfrak{m}, \text{TA}_i) = \text{I } \mathfrak{m} (\overline{\text{I}} \text{ x}) \{ \dots \}$ is derivable, for all TA_i such that $\mathfrak{m} \in \text{meth}(\text{TA}_i)$, by Lemma 3. To conclude, we know that $\text{altlook}(\cdot, \text{TA}_i)$ is a function. Which ensures they are all equal. \square*

In the following, we suppose that Theorem 1 holds, and we can address *mbody* and *mtype* as mathematical functions. We prove that *msig* has the same semantics than the **requires** set of trait checking. Both are expected to give the signatures of the methods which are required in a trait alteration.

Lemma 8 (Require Soundness)

If $\text{TA OK requires } \overline{\text{S}}$, then $\text{msig}(\mathfrak{m}, \text{TA}) = \text{S}$ if and only if $\text{S} \in \overline{\text{S}}$.

Proof *Straightforward induction on the derivation of $\text{msig}(\mathfrak{m}, \text{TA}) = \text{S}$. Here are the most relevant cases:*

- (MSig·Ali₂) *In both rules (Alias·Ok₁) and (Alias·Ok₂), it is easy to observe that the new name of the method is not in the **requires** list (the role of (Alias·Ok₂) is actually to remove it from the **requires** list if it appears in the previous step). Hence the result.*
- (MSig·Ali₃) *In both rules (Alias·Ok₁) and (Alias·Ok₂), $\mathfrak{n} \in \text{meth}(\text{TA}_i)$ is a precondition. By Lemma 4, it contradicts the precondition of rule (MSig·Ali₃). This case never occurs when a trait alteration typechecks. \square*

We prove that trait alterations do not alter the type interface of a trait.

Lemma 9 (head Soundness)

If $\text{TA OK requires } \overline{\text{S}}$ and $\text{mtype}(\mathfrak{m}, \text{head}(\text{TA})) = \overline{\text{I}} \rightarrow \text{I}$, then $\text{mtype}(\mathfrak{m}, \text{TA}) = \overline{\text{I}} \rightarrow \text{I}$.

Proof *By induction on TA . Let us prove the most significant cases*

- *($\text{TA} = \text{TA}_1 \text{ minus } \{\mathfrak{p}\}$), with $\mathfrak{p} \neq \mathfrak{m}$. Then, $\text{mtype}(\mathfrak{m}, \text{TA})$ is inferred from either $\text{altlook}(\mathfrak{m}, \text{TA})$ or $\text{msig}(\mathfrak{m}, \text{TA})$ (thanks to (MTyp·Impl) or (MTyp·Virt), respectively). In both cases the result is straightforward.*
- *($\text{TA} = \text{TA}_1 \text{ minus } \{\mathfrak{m}\}$). Since TA typechecks, then $\text{altlook}(\mathfrak{m}, \text{TA}_1) \neq \text{fail}$, by Lemma 4. We also have, by definition of altlook , that $\text{altlook}(\mathfrak{m}, \text{TA}) = \text{fail}$. By definition of mtype , we have that $\text{mtype}(\mathfrak{m}, \text{TA})$ is inferred from $\text{msig}(\mathfrak{m}, \text{TA})$ (rule (MTyp·Virt)), and that $\text{mtype}(\mathfrak{m}, \text{TA}_1)$ is inferred from $\text{altlook}(\mathfrak{m}, \text{TA}_1)$ (rule (MTyp·Impl)). By rule (MSig·Ex₁), we have $\text{mtype}(\mathfrak{m}, \text{TA}_1) = \overline{\text{I}} \rightarrow \text{I}$*

implies $\text{mtype}(\mathbf{m}, \mathbf{TA}) = \bar{\mathbf{I}} \rightarrow \mathbf{I}$. By induction hypothesis we have the result.

- ($\mathbf{TA} = \mathbf{TA}_1$ with $\{\mathbf{n}@m\}$). Since \mathbf{TA} typechecks, we can derive that \mathbf{TA}_1 typechecks and $\text{altlook}(\mathbf{m}, \mathbf{TA}_1) = \text{fail}$. Then, we know that if $\text{mtype}(\mathbf{m}, \mathbf{TA}_1) = \bar{\mathbf{I}} \rightarrow \mathbf{I}$, then $\text{msig}(\mathbf{m}, \mathbf{TA}_1) = \mathbf{I} \ \mathbf{m}(\bar{\mathbf{I}} \ \bar{\mathbf{x}})$. By Lemma 8, it follows that $\mathbf{I} \ \mathbf{m}(\bar{\mathbf{I}} \ \bar{\mathbf{x}}) \in \bar{\mathbf{S}}'$ (where \mathbf{TA}_1 OK requires $\bar{\mathbf{S}}'$). We can deduce that \mathbf{TA} is typechecked through the (Alias·Ok₂) rule. It is then obvious that $\text{mtype}(\mathbf{m}, \mathbf{TA}_1) = \bar{\mathbf{I}} \rightarrow \mathbf{I}$ implies $\text{mtype}(\mathbf{m}, \mathbf{TA}) = \bar{\mathbf{I}} \rightarrow \mathbf{I}$ (it is enforced directly by the (Alias·Ok₂) rule). By induction hypothesis, we have the result. \square

From now on the lemma and theorem sequence is the same as in FJ and FTJ.

Lemma 10 (mtype Soundness)

If $\text{mtype}(\mathbf{m}, \mathbf{J}) = \bar{\mathbf{L}} \rightarrow \mathbf{L}$, then $\text{mtype}(\mathbf{m}, \mathbf{I}) = \bar{\mathbf{L}} \rightarrow \mathbf{L}$, for all $\mathbf{I} <: \mathbf{J}$.

Proof Straightforward induction on the derivation of $\mathbf{I} <: \mathbf{J}$, using Lemma 9 for the cases (Sub·Tr) and (Sub·Cla·Tr). We show the most difficult cases

- (Sub·Cla) Let's assume that $\text{mtype}(\mathbf{m}, \mathbf{D}) = \bar{\mathbf{L}} \rightarrow \mathbf{L}$. We want to prove that $\text{mtype}(\mathbf{m}, \mathbf{C}) = \bar{\mathbf{L}} \rightarrow \mathbf{L}$. It can be achieved by several rules.
 - (MTyp·Self) Then, the result is obtained thanks to the rule (Mth·Ok·Cla); in particular, thanks to the override clause in it.
 - (MTyp·Tr) Let us suppose that the type of \mathbf{m} is obtained from the trait \mathbf{TA}_i . We then know that $\text{sig}(\mathbf{TA}_i)$ OK IN \mathbf{C} . In particular, for any $\mathbf{I} \ \mathbf{m}(\bar{\mathbf{I}} \ \bar{\mathbf{x}}) \in \text{sig}(\mathbf{TA}_i)$ we know that $\text{override}(\mathbf{m}, \mathbf{D}, \bar{\mathbf{I}} \rightarrow \mathbf{I})$ holds. We can rewrite it thanks to Lemma 6 to if $\text{mtype}(\mathbf{m}, \mathbf{D}) = \bar{\mathbf{I}} \rightarrow \mathbf{I}$, then $\text{mtype}(\mathbf{m}, \mathbf{TA}_i) = \bar{\mathbf{I}} \rightarrow \mathbf{I}$.
 - (MTyp·Super) This case is direct. \square

Lemma 11 (Substitution lemma)

If $\Gamma, \bar{\mathbf{x}}:\bar{\mathbf{J}} \vdash \mathbf{e} \in \mathbf{J}$ and $\Gamma \vdash \bar{\mathbf{d}} \in \bar{\mathbf{I}}$, where $\bar{\mathbf{I}} <: \bar{\mathbf{J}}$, then $\Gamma \vdash [\bar{\mathbf{d}}/\bar{\mathbf{x}}]\mathbf{e} \in \mathbf{I}$ for some $\mathbf{I} <: \mathbf{J}$.

Proof By induction on the derivation of $\Gamma, \bar{\mathbf{x}}:\bar{\mathbf{J}} \vdash \mathbf{e} \in \mathbf{J}$. We will show only two cases the other ones are straightforward

- (Typ·Call). By induction hypothesis, we have $\Gamma \vdash [\bar{\mathbf{d}}/\bar{\mathbf{x}}]\mathbf{e}_0 \in \mathbf{J}_0$, and $\mathbf{J}_0 <: \mathbf{I}_0$, and $\Gamma \vdash [\bar{\mathbf{d}}/\bar{\mathbf{x}}]\mathbf{e} \in \bar{\mathbf{I}}'$, and $\bar{\mathbf{I}}' <: \bar{\mathbf{I}}$. Then, by Lemma 10, we have that $\text{mtype}(\mathbf{m}, \mathbf{J}_0) = \text{mtype}(\mathbf{m}, \mathbf{I}_0) = \bar{\mathbf{J}} \rightarrow \mathbf{I}$. By transitivity we also have $\bar{\mathbf{I}}' <: \bar{\mathbf{J}}$. Then, we conclude that $\Gamma \vdash \mathbf{e}_0.\mathbf{m}(\bar{\mathbf{e}}) \in \mathbf{I}$ (and obviously $\mathbf{I} <: \mathbf{J}$).
- (Typ·Field). By induction hypothesis, we have $\Gamma \vdash [\bar{\mathbf{d}}/\bar{\mathbf{x}}]\mathbf{e}_0 \in \mathbf{I}_0$ for some $\mathbf{I}_0 <: \mathbf{C}_0$. An easy induction on the derivation of $\mathbf{I}_0 <: \mathbf{C}_0$ shows that $\mathbf{I}_0 = \mathbf{D}_0$ for some \mathbf{D}_0 . Then, it is easy to show that $\mathbf{D}_0 <: \mathbf{C}_0$ implies $\text{fields}(\mathbf{D}_0) \subseteq \text{fields}(\mathbf{C}_0)$. Hence the result. \square

Lemma 12 (Weakening)

If $\Gamma \vdash \mathbf{e} \in \mathbf{C}$, then $\Gamma, \mathbf{x}:\mathbf{D} \vdash \mathbf{e} \in \mathbf{C}$.

Proof Straightforward induction. \square

Lemma 13 (Method Body Type)

If $\text{mtype}(\mathbf{m}, \mathbf{C}) = \bar{\mathbf{J}} \rightarrow \mathbf{J}$ and $\text{mbody}(\mathbf{m}, \mathbf{C}) = (\bar{\mathbf{x}}, \mathbf{e})$, then for some \mathbf{I}_0 with $\mathbf{C} <: \mathbf{I}_0$,

there exists $I <: J$ such that $\bar{x}:\bar{J}, \text{this}:I_0 \vdash e \in I$.

Proof We prove the following statement by induction on the derivation of $\text{mbody}(m, C) = (\bar{x}, e)$: if $\text{mbody}(m, C) = (\bar{x}, e)$, then $J \text{ m } (\bar{J} \bar{x}) \{\text{return } e; \} \text{ OK IN } I_0$ for some I_0 with $C <: I_0$ and some $\bar{J} \rightarrow J$. Then, by Lemma 10, $\bar{J} \rightarrow J = \text{mtype}(m, C)$ holds.

- (MBdy·Cla) Immediate from (Cla·Ok) rule.
- (MBdy·Tr) Straightforward induction on the derivation of $\text{altlook}(m, \text{TA}) = J \text{ m } (\bar{J} \bar{x}) \{\text{return } e; \}$.
- (MBdy·SCla) Induction case. \square

We are ready to prove the main theorems.

Theorem 2 (Subject Reduction)

If $\Gamma \vdash e \in J$ and $e \rightarrow e'$, then $\Gamma \vdash e' \in I$, for some $I <: J$.

Proof We prove it by a straightforward induction on the derivation of $e \rightarrow e'$. The base case (reduction of the head redex) is done by a straightforward case analysis on the reduction rule used. This proof has no difficult content, however if a reader is interested, a comprehensive account of the details can be found in the original FJ paper [9]. We provided all the lemmas used in the proof, so it works also for iFTJ (the key lemma being Lemma 13). \square

Theorem 3 (Progress)

Suppose e is a well-typed expression.

- (1) If e includes $\text{new } C(\bar{e}).f$ as a subexpression, then $\text{fields}(C) = \bar{T} \bar{f}$ and $f \in \bar{f}$.
- (2) If e includes $\text{new } C(\bar{e}).m(\bar{f})$ as a subexpression, then $\text{mbody}(m, C) = (\bar{x}, e_0)$ and $\#(\bar{x}) = \#(\bar{d})$.

Proof The proof is straightforward: subexpression are well-typed, thus we can assume that the subexpression appears at the head of e . Then, the result is deduced directly from the typing rules. Theorem 1 is essential for this proof. \square

Theorem 4 (Reduction preserves safety)

If e is safe in Γ , and $e \rightarrow e'$, then e' is safe in Γ .

Proof This proof is just similar to the Subject Reduction proof. \square

Theorem 5 (Progress of safe programs)

Suppose e is safe in Γ . If e has $(C)\text{new } D(\bar{e})$ as a subexpression, then $D <: C$.

Proof The result is straightforward from the definition of safety. Indeed the only rule that can be applied to derive the type of $(C)\text{new } C_0(\bar{e})$ is (Typ·UCast). The result follows directly. \square